ROBotic Open-architecture Technology for
Cognition, Understanding and Behavior

Project No. 004370

RobotCub
Development of a Cognitive Humanoid Cub

Instrument:	Integrated Project
Thematic Priority:	IST - Cognitive Systems

D8.2 Definition of Documentation and Manufacturing Procedures

Due date: **1/09/2005**
Submission Date: **19/09/2005**

Start date of project: **01/09/2004** Duration: **60 months**

Organisation name of lead contractor for this deliverable: **University of Genoa**

Responsible Person: **F. Becchi, G. Metta, D. Vernon**

Revision: **1.0**

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	**PU**
PP	Restricted to other programme participants (including the Commission Service)	
RE	Restricted to a group specified by the consortium (including the Commission Service)	
CO	Confidential, only for members of the consortium (including the Commission Service)	

Contents

Overview

Deliverable D8.2 sets out the procedures by which the iCub is to be manufactured and documented, addressing both the hardware mechatronic components and the software systems. It comprises three major parts: the first deals with the procedures for manufacturing and documenting the *iCub* mechatronic platform, the second comprises a set of guidelines on downloading and uploading *iCub* software from and to the CVS repository, and the third provides a set of guidelines for software documentation, programming style, and programming practice. The third part also provides guidance on how to name and structure files to be included in the *iCub* software repository. The intention is that these guidelines should be short enough to be easily adopted and applied, but long enough to be useful in the creation of high-quality easily-maintained software.

Part I
Mechanical and Electronic Design Rules

1 Introduction

The basis for collaboration between the different groups involved in the design of the *iCub* platform is the definition of common rules and tools for the design itself. From the very beginning several instances were fixed in order to maximize the information interchange between the partners and the integration of all the contributions in a single final design. In this part, the standards for the mechanical documentation are set out. Detailed CAD standards are defined, some of which are platform-dependent. A brief survey of the different manufacturing technologies and procedures adopted for the realization of the first prototypes of the *iCub* is also included.

2 Documentation standards and procedures

The *RobotCub* consortium has chosen PTC/Pro ENGINEER Wildfire 2.0 as the CAD software package for all official *iCub* design work. This choice of commercial sofware was necessary as no open software package capable of dealing with a system of the complexity of *iCub* was available.

2.1 Mechanical Documentation Coding Standards

This section explains the coding standards agreed for the mechanical documentation. All the design development done in the *RobotCub* project must adhere to these standards to ensure complete traceability of the file. The same coding is also recommended as an internal standard for each group involved in the *iCub* mechanical platform documentation.

The coding procedure gives an alphanumeric filename code to each file produced in the design process from part files to assembly drawings. The same coding should also be used for the written documentation (*e.g.* calculation report).

The fields in the code are:

`RC` common to all Robotcub Consortium documentation.

`GROUP`: this denotes the acronym of the group to which the author belongs. Currently defined acronyms are listed below:

UGDIST	SSSA
UNIZH	UNIUP
UNIFE	UNIHER
IST	UNISAL
EPFL	TLR
EBRI	

VERSION: this is a numeric field unique for each different design or solution. This is the main assembly code: it allows for different designs of the same assembly. For example, there might be several different (*i.e.* alternative) shoulder assemblies and each design will have its own unique version number.

LETTER: The letter is a type identifier which signifies what the file contains, as follows.

A	Top Assembly (the top assembly is the higher assembly in the model tree)
G	Group (a group is a lower level assembly in the model tree)
P	Part
D	Documentation file (*e.g.* doc or xls files
LY	2D layout file (*e.g.* preliminary 2D design on Autocad)

CODE: This is the identification code in the current project (each part or group has its own identification code).

RELEASE: this is the release number of the file (this code must match the REVISION in the drawing; see later).

DESCRIPTION: this is an extra field that can be used for better file identification (*e.g.* `left eye pulley`).

Following this standard for naming files, the first part of the first design made by Telerobot would be:

```
RC_TLR_001_P_001_00_FIRST_PART
```

Note: No spaces are to be included in the filename. Use the underscore "_" to separate words if necessary.

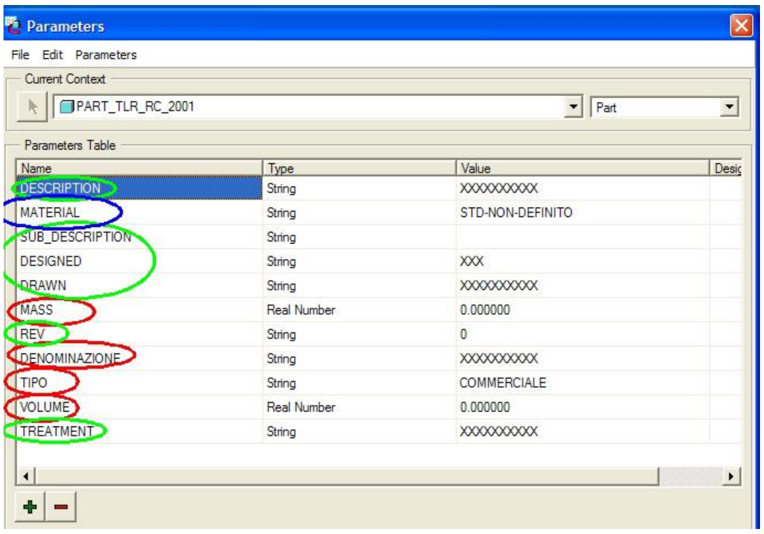
2.2 CAD Documentation Standards

A set of rules for the correct definition of each file of a 3D CAD Pro/ENGINEER model is presented in this section. Standard ROBOTCUB part (`*.prt`) and standard ROBOTCUB assembly (`*.asm`) are defined. The drawing template and consequent parameters is also discussed.

2.2.1 Standard Part

The standard part for *RobotCub* project is named "`PART_RC_2001.prt`". This part must be used for creating new ones (using the "`Save as ...`" or the "`New`" file options commands). For common reference, the standard part is the same for all the consortium and must not be modified. The parameters defined in each part file are listed below.

`DESCRIPTION`	Description of the part
`SUBDESCRIPTION`	Optional sub-description for the part (*i.e.* the group in which this part will be assembled, . . .)
`DESIGNED`	The group acronym (see above)
`DRAWN`	The name of the group designer
`REV`	Revision of the part (numeric: 1, 2, 3, ...)
`TREATMENT`	The treatment that will be applied to the part
`MATERIAL`	The material assigned to the part. This parameter is filled in using the Pro/E command "Setup – Material– Assign – From Part".



PART_TLR_RC_2001 — Part

Name	Type	Value	Desig
DESCRIPTION	String	XXXXXXXXXX	
MATERIAL	String	STD-NON-DEFINITO	
SUB_DESCRIPTION	String		
DESIGNED	String	XXX	
DRAWN	String	XXXXXXXXXX	
MASS	Real Number	0.000000	
REV	String	0	
DENOMINAZIONE	String	XXXXXXXXXX	
TIPO	String	COMMERCIALE	
VOLUME	Real Number	0.000000	
TREATMENT	String	XXXXXXXXXX	

The parameters highlighted in green are filled by the user.

The materials that can be used are already reported in the standard part. The list is defined from previous similar design experience. The list can be enlarged if required. This procedure is necessary in order to assign the right density to the part (and consequently the right mass). The default material assigned to the standard part is `Al6082`. The right one must be assigned. The materials added to the standard part are as follows.

`17-4PH`	`Al6082`
`Armonic steel`	`Brass`
`Carbon fiber`	`Delrin`
`Ergal70`	`Lexan`
`Neoprene`	`Orkot`
`PE`	`PEEK`
`Plexiglass`	`PP`
`PTFE`	`PVC`
`Rynite 555`	`AISI 316`
`AISI 420`	

The parameters highlighted in red:

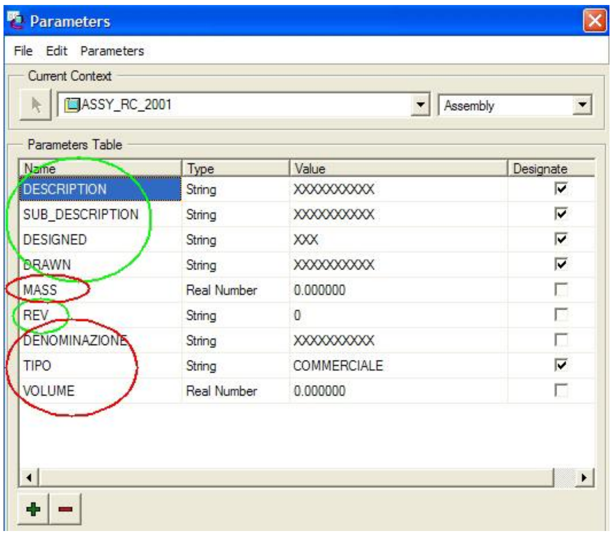
```
MASS
VOLUME
DENOMINAZIONE
TIPO
```

are filled by the system at the first regeneration of the part. `MASS` is directly dependent for the `MATERIAL` parameter (using the density value reported in the material file).

2.2.2 Standard Assembly

The standard assembly for the *RobotCub* project is named “`ASSY_RC_2001.asm`”. This file must be used when creating a new assembly (using the “Save as . . .” or the “New” file options commands).

In the same way as the standard part, for consistency the standard assembly is the same for all the consortium and must not be modified. The parameters defined in each assembly file are listed below.



Current Context: ASSY_RC_2001 — Assembly

Parameters Table

Name	Type	Value	Designate
DESCRIPTION	String	XXXXXXXXXX	✓
SUB_DESCRIPTION	String	XXXXXXXXXX	✓
DESIGNED	String	XXX	✓
DRAWN	String	XXXXXXXXXX	✓
MASS	Real Number	0.000000	☐
REV	String	0	☐
DENOMINAZIONE	String	XXXXXXXXXX	☐
TIPO	String	COMMERCIALE	✓
VOLUME	Real Number	0.000000	☐

As described before, the parameter highlighted in green are defined by the user. The red ones are defined by the system at the first regeneration. Note: the `MASS` parameter is automatically calculated by the system as a sum of all the parts mounted in assembly in every regeneration of the model.

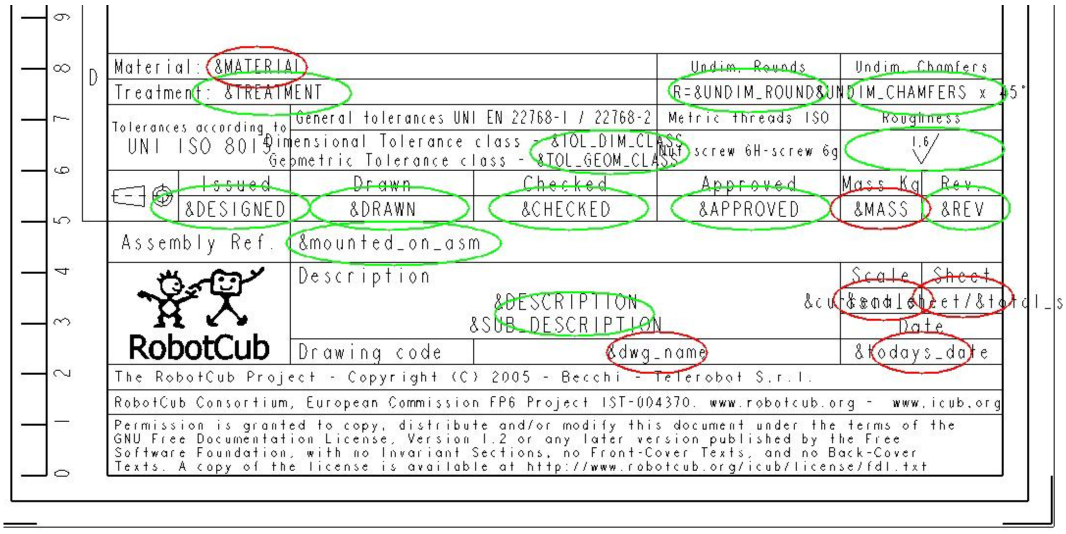
2.2.3 Drawings — General

The standard sheet drawing format are:

`Robot_cub_a0_mech`	A0 format for mechanical drawings
`Robot_cub_a0_ass`	A0 format for assembly drawings
`Robot_cub_a1_mech`	A1 format for mechanical drawings
`Robot_cub_a1_ass`	A1 format for assembly drawings
`Robot_cub_a2_mech`	A2 format for mechanical drawings
`Robot_cub_a2_ass`	A2 format for assembly drawings
`Robot_cub_a3_mech`	A3 format for mechanical drawings
`Robot_cub_a3_ass`	A3 format for assembly drawings
`Robot_cub_a4_mech`	A4 format for mechanical drawings
`Robot_cub_a4_ass`	A4 format for assembly drawings

2.2.4 Mechanical Drawings Template

The main table included in the mechanical drawings is described below.


Material: &MATERIAL
Treatment: &TREATMENT
Undim. Rounds
R=&UNDIM_ROUND
Undim. Chamfers
&UNDIM_CHAMFERS x 45°
Tolerances according to
UNI ISO 8015
General tolerances UNI EN 22768-1 / 22768-2
Dimensional Tolerance class - &TOL_DIM_CLASS
Geometric Tolerance class - &TOL_GEOM_CLASS
Metric threads ISO
Nut screw 6H-screw 6g
Roughness
1.6
Issued
&DESIGNED
Drawn
&DRAWN
Checked
&CHECKED
Approved
&APPROVED
Mass Kg
&MASS
Rev.
&REV
Assembly Ref.
&mounted_on_asm
RobotCub
Description
&DESCRIPTION
&SUB_DESCRIPTION
Scale
&scale
Sheet
&current_sheet/&total_s
Date
&todays_date
Drawing code
&dwg_name
The RobotCub Project - Copyright (C) 2005 - Becchi - Telerobot S.r.l.
RobotCub Consortium, European Commission FP6 Project IST-004370. www.robotcub.org - www.icub.org
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation, with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is available at http://www.robotcub.org/icub/license/fdl.txt

The values highlighted in red are automatically filled in by the system:

`MATERIAL`	The material assigned to the part (see above)
`DWG_NAME`	The name of the drawing file; this must be the same as the reference 3D model
`SCALE`	Scale value of the drawing
`SHEET`	Sheet number (*i.e.* for two sheets of the same drawing: 1/2, 2/2, ...)
`DATE`	Drawing creation date
`MASS`	Mass value (part parameter, see above)

The values highlighted in green have are defined by the user if not already filled in the reference part:

`TREATMENT`	Treatment assigned to the part (if completed in the part, the value will be reported in the table automatically)
`UNDIM. ROUNDS`	Undimensioned rounds in the part
`UNDIM CHAMFERS`	Undimensioned chamfers in the part
`ISSUED`	The company abbreviation in *RobotCub* (*i.e.* for Telerobot is TLR) (already completed in the part)
`DRAWN`	The name of the group designer
`CHECKED`	The name of the group checker
`APPROVED`	The name of the group designer
`REV.`	Revision of the document (part parameter, see above)
`DIMENSIONAL TOLERANCE CLASS`	Dimensional tolerance class; see below for more details
`GEOMETRIC TOLERANCE CLASS`	Geometric tolerance class; see below for more details
`ROUGHNESS`	Part roughness; see below for more details
`ASSEMBLY REF.`	Assembly in which the part is mounted (assembly code); the parameter name is "mounted on asm"
`DESCRIPTION`	Description and eventually sub-description of the part (already completed in the part)

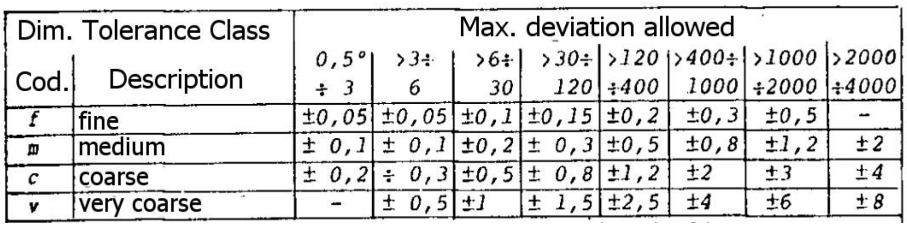
When the user inserts a format in the drawing, the system automatically requires the `TYPE` of all the parameter which aren't in the reference part (that is to say: `UNDIM. ROUNDS`, `UNDIM CHAMFERS`, `CHECKED`, `APPROVED`, `REV.`, `MOUNTED_ON_ASM`, `TOL_DIM_CLASS`, `TL_GEOM_CLASS`). For all the parameters, the type is "`STRING`". After the type definition, the user has to fill the parameter value for every parameter requested. Note: parameters can be easily modified with the "`Edit - Value`" command.

2.2.5 Tolerance Classes and Roughness

For Dimensional tolerance class parameter, the following values have to be used:

```
f
m
c
v
```

Note: these values have to be written in *lower* case.

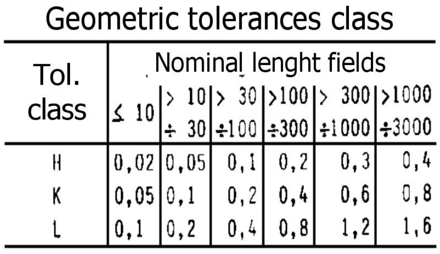


Dim. Tolerance Class		Max. deviation allowed							
Cod.	Description	0,5° ÷ 3	>3÷ 6	>6÷ 30	>30÷ 120	>120 ÷400	>400÷ 1000	>1000 ÷2000	>2000 ÷4000
f	fine	±0,05	±0,05	±0,1	±0,15	±0,2	±0,3	±0,5	–
m	medium	± 0,1	± 0,1	±0,2	± 0,3	±0,5	±0,8	±1,2	±2
c	coarse	± 0,2	÷ 0,3	±0,5	± 0,8	±1,2	±2	±3	±4
v	very coarse	–	± 0,5	±1	± 1,5	±2,5	±4	±6	±8

For Geometric tolerance class parameter, the values that have to be used are the following, according to the table below reported:

```
H
K
L
```

Note: these values have to be written in *upper* case.

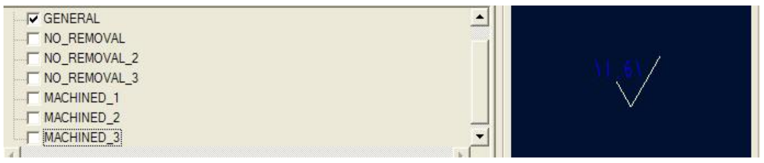


Geometric tolerances class

Tol. class	Nominal lenght fields					
	≤ 10	> 10 ÷ 30	> 30 ÷100	>100 ÷300	> 300 ÷1000	>1000 ÷3000
H	0,02	0,05	0,1	0,2	0,3	0,4
K	0,05	0,1	0,2	0,4	0,6	0,8
L	0,1	0,2	0,4	0,8	1,2	1,6

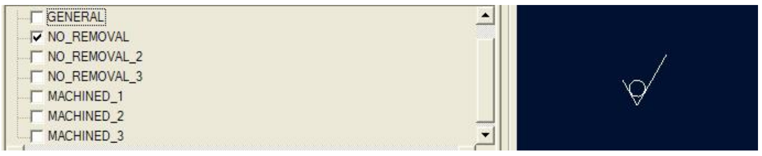
The roughness reported in the drawing format is a symbol chosen from the options below.

General



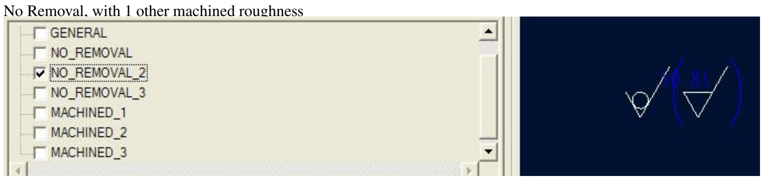
[✓ GENERAL]
NO_REMOVAL
NO_REMOVAL_2
NO_REMOVAL_3
MACHINED_1
MACHINED_2
MACHINED_3

Only No Removal



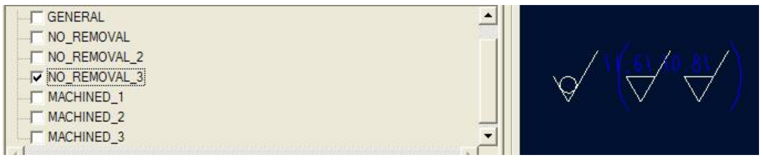
GENERAL
[✓ NO_REMOVAL]
NO_REMOVAL_2
NO_REMOVAL_3
MACHINED_1
MACHINED_2
MACHINED_3

No Removal, with 1 other machined roughness



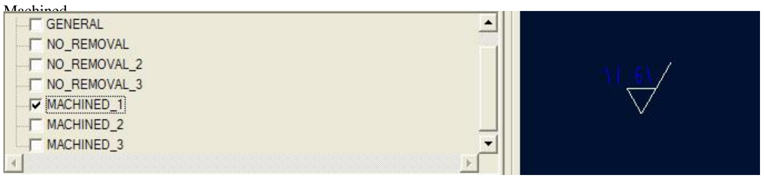
GENERAL
NO_REMOVAL
[✓ NO_REMOVAL_2]
NO_REMOVAL_3
MACHINED_1
MACHINED_2
MACHINED_3

No Removal, with 2 other machined roughnesses



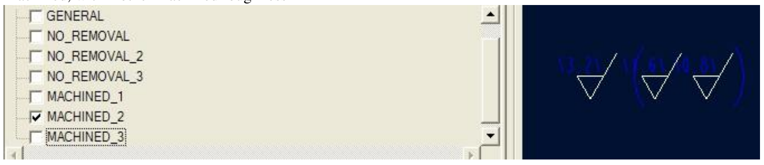
GENERAL
NO_REMOVAL
NO_REMOVAL_2
[✓ NO_REMOVAL_3]
MACHINED_1
MACHINED_2
MACHINED_3

Machined



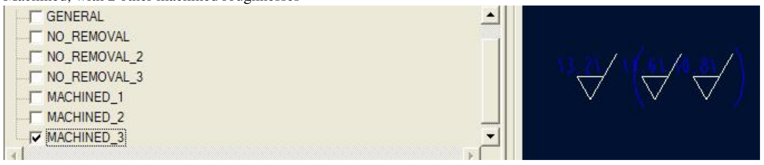
GENERAL
NO_REMOVAL
NO_REMOVAL_2
NO_REMOVAL_3
[✓ MACHINED_1]
MACHINED_2
MACHINED_3

Machined, with 1 other machined roughness



GENERAL
NO_REMOVAL
NO_REMOVAL_2
NO_REMOVAL_3
MACHINED_1
[✓ MACHINED_2]
MACHINED_3

Machined, with 2 other machined roughnesses

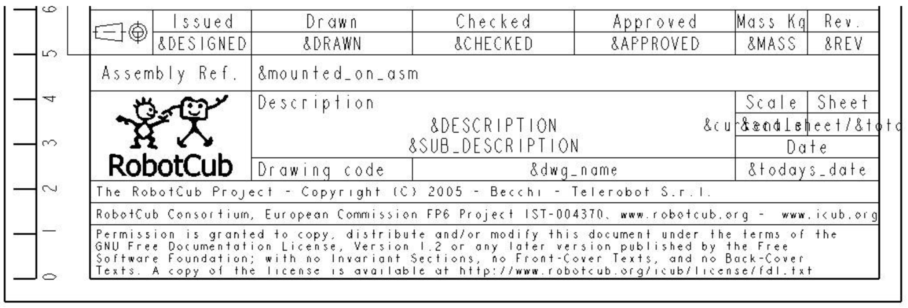


GENERAL
NO_REMOVAL
NO_REMOVAL_2
NO_REMOVAL_3
MACHINED_1
MACHINED_2
[✓ MACHINED_3]

In all cases the symbol must be completed before its insertion with the "`Var text`" value. For roughness symbol in the drawing, the Pro/ENGINEER standard symbol ("`isosurftext.sym`") in the "`System Sym`" directory should be used.

2.2.6 Assembly Drawings Template

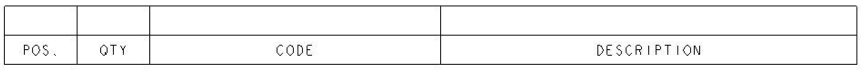
The main table reported in assembly drawing formats is described here.



	Issued	Drawn	Checked	Approved	Mass Kg	Rev.
	&DESIGNED	&DRAWN	&CHECKED	&APPROVED	&MASS	&REV
Assembly Ref.	&mounted_on_asm					
RobotCub	Description				Scale	Sheet
	&DESCRIPTION &SUB_DESCRIPTION				&scale	&cu[illegible]heet/&tot
					Date	
	Drawing code	&dwg_name			&todays_date	
The RobotCub Project - Copyright (C) 2005 - Becchi - Telerobot S.r.l.						
RobotCub Consortium, European Commission FP6 Project IST-004370, www.robotcub.org - www.icub.org						
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is available at http://www.robotcub.org/icub/license/fdl.txt						

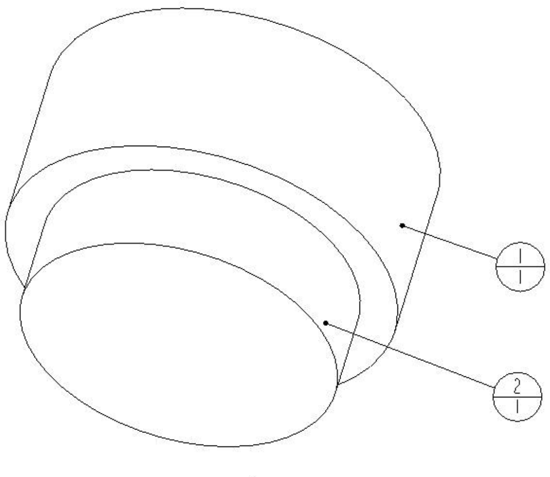
It is similar to the mechanical drawing table, without the last 4 rows. The same instructions apply.

In case of an assembly drawing, it might be necessary the insert a BOM (Bill Of Material) table. The "`BOM_table`" file can be used. This table will be automatically filled by the system, reading information reported in parts (or sub-assemblies) parameters.



POS.	QTY	CODE	DESCRIPTION

The "with qty" `BOM` balloon type can be used. An image of a `BOM` ballooned assembly is shown below.



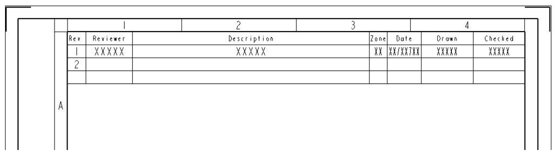
2	1	B	PART "B"
1	1	A	PART "A"
POS.	QTY	CODE	DESCRIPTION

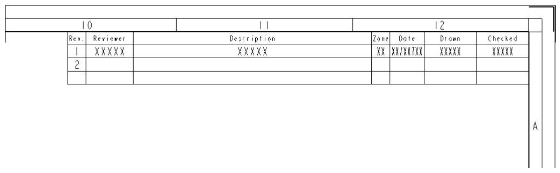
2.2.7 Revision Tables

In the case of a document revision, the rules below described will apply.

Insert the new revision table (in the upper part of the drawing for A4, the upper-right for the other formats; see diagram below).

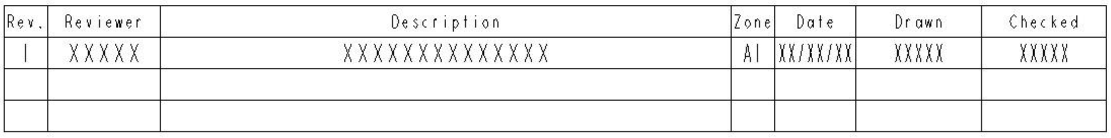
`Rev1_table`	for the first revision on the document
`Rev2_table`	for the second revision on the document
`Rev3_table`	for the third revision on the document





The required parameters are as follows (always `STRING` type).

`Reviewer`	the group that creates the document revision
`Description`	revision description
`Zone`	the zone in the drawing where this revision is visible
`Date`	the revision date
`Drawn`	who draws the revision
`Checked`	who checks the revisioned document



Rev.	Reviewer	Description	Zone	Date	Drawn	Checked
1	XXXXX	XXXXXXXXXXXXXX	A1	XX/XX/XX	XXXXX	XXXXX

The same instructions must be followed in case of a second or third revision (the table with the previous revision has to be deleted).

3 Manufacturing Procedures

In the design of the *iCub* mechanical components, several different technologies were discussed and analyzed. To date, the prototypes have have typically been realized using:

- Aluminium alloys/steel machining (with traditional CNC mill, lathe, laser cut, spark erosion and wire cut);
- Rapid prototyping for preliminary design evaluation or as starting point for other manufacturing technology (as for external cover made with glass or carbon fiber starting from a RP model).

The experience of each group has been shared by all the consortium in the definition of best design approach. No defined procedures have yet been agreed. Once the first complete prototype has been integrated and tested, industrial drawing and manufacturing, assembly, and integration procedures will be defined.

Part II
The *iCub* CVS Repository

4 Getting started with the iCub

The iCub software is maintained by means of CVS. The CVS repository can be accessed at: "`cvs.robotcub.org`" or browsed on the web at: "`http://www.robotcub.org/iCub`".
The access to the repository is only by secure shell (ssh). Before accessing the repository you need to ask the maintainer (LIRA-Lab at the moment of writing) to create an account for you and set your password. The ssh access allows full control of the repository (add, checkout, commit).
The repository access has been tested also from the popular WinCVS client. It delivers reasonable performances. If you're not keen on GUIs you can still use cvs from the command line both on Linux and Windows systems.
The same code can be accessed also from SourceForge. We have a project called "`robotcub`" hosted in SourceForge. The project group page is at:

```
http://sf.net/projects/robotcub
```

while the home page:

```
http://robotcub.sf.net
```

redirects you to the standard RobotCub.org website.
Once you receive the account name and password you can log in with any ssh client and use "`passwd`" to change your password to something you like. You should also start setting up your environment to download the code to your local machine. CVS, in short, works by creating a local copy of the repository on your local hard drive that you can use to modify and compile the code. Changes can later be uploaded back to the common repository through the "`commit`" command.
The remote repository location and connection method is specified by an environment variable called CVSROOT. The following two lines:

```
export CVSROOT=:ext:YOURNAME@cvs.robotcub.org:/cvsroot/robotcub
export CVS_RSH=/usr/bin/ssh
```

would set the CVSROOT to use ssh and connect to cvs.robotcub.org with username "`YOURNAME`" and sets the root to "`/cvsroot/robotcub`". Note that these are exactly the pathnames you should use to connect to the iCub repository. On Linux, you can simply change to the directory you intend to use for your local copy of the repository and type:

```
$ cvs co -P iCub
```

which checks the "`iCub`" module out and prune (-P) the empty directories in the repository (does not check out the empty directories). On Windows you should setup things a bit differently. A tutorial on the CVS client configuration is available for example from:

```
http://sourceforge.net/docman/display_doc.php?docid=25888&group_id=1
```

You can also use the *cvsnt* package from the command line, e.g:

```
D:\Users\pasa\Repository\prova>cvs -d:ssh;username=pasa;hostname=cvs.robot
cub.org:/cvsroot/robotcub co iCub
Password:
Response: **********
cvs checkout: cwd=D:\Users\pasa\Repository\iCub ,current=D:\Users\pasa\Rep
ository\iCub
cvs server: Updating iCub
U iCub/AUTHORS
U iCub/COPYING
U iCub/INSTALL
U iCub/README
U iCub/TODO
cvs server: Updating iCub/bin
U iCub/bin/dummy.txt
```

and so forth. The command above asks for a checkout (co), uses ssh for authentication with unsername set to *pasa* and hostname *cvs.robotcub.org* and root directory */cvsroot/robotcub*. Finally, it asks for the module *iCub* which is the only available module at the moment of writing.
The basic commands available to CVS for manipulating the repository are:

- add: to add files and directories to the repository
- update: to get an up-to-date version of the code from the repository
- checkout: to get the first-time copy (or a new one) of the repository
- commit: to upload changes to any file into the repository

Please consult the CVS manual (or help) for more information about using CVS. It is worth noting that CVS does not allow removing or renaming directories, thus extreme care should be taken when creating new directories. The administrator can move or rename directories but it has to be done manually and it can risk the integrity of the repository (depending on the administrator of course). CVS was chosen for its stability. It is an "old" application which is now fairly stable and well tested. SourceForge uses CVS and, although there are new versioning applications being developed, they are not yet as stable as CVS. These were the reasons for choosing CVS for serving our software repository. The simpler the better.
Finally, CVS allows concurrent development of code by merging changes to files. This mechanism has to be used parsimoniously but it works well in general. Versions are maintained of every file so that older versions can be recovered at any time. In the following, the forward slash / is always used even to indicate Windows paths; depending on the shell/terminal you use you might need to replace the forward with the backslash.

5 Software structure

The software directory tree was chosen to be modular and extensible. The idea is to live with many separate modules perhaps developed by different people that glued together will make the iCub to function. Although CVS allows the creation of various independent modules they cannot be organized hierarchically, which makes things a bit more difficult. It has been chosen then to define modules simply by the placing them in different directories without relying on the CVS module mechanism. The main module of our repository is called "`iCub`" which is also the root directory on the repository.

When checking out the code you're are requested to give a module name. This is exactly "`iCub`". The capitalized "C" is mandatory.
The following table summarizes the directory organization:

Directory name	Description
conf	configuration and template files
doc	documentation and manuals
license	license files and templates
src	the source code

The "`src`" directory is the main container of the source code. It contains various subdirectories which contain in turn independent modules (logically distinct pieces of software). Each subdirectory can be organized freely by the developers as long as it compiles properly with the rest of the system. In organizing the code development, Makefile's are provided and rules given in a tentative to harmonize the final result.
The filenames should be consistent with the RobotCub standards in:

`RC_DIST_DV_standards_filename_convention.pdf`

that can be found in the "`license`" directory. The iCub header files, once intalled, should be eventually contained all into a common directory. It is perhaps best to avoid mingling with the standard system header files. For this reason header files are included always from a subdirectory called (surprise surprise!) "`iCub`" of any include directory. An example of inclusion in C or C++ would be something like this:

```
#include <iCub/RC_DIST_GM_cognition_module.h>
```

Thus, you should organize your header files in something like "`include/iCub`" and provide a path to the compiler of the form "`include`". Installation can copy these files somewhere, as for example, into "`/usr/local/include/iCub`". Makefile's are provided which would take care of installing things properly (if used). Standard templates can be found in "`conf`". An example of a module is the "`logpolar`" described next.
We distinguish three types of compilation and/or installation that perform different operations:

- **local**: used when developing on the current module;
- **install**: used when developing across many modules;
- **system install**: used to install the iCub code for general use on a given machine.

Thus when developing a specific module, the user does not have to disturb the rest of the system, unless there are dependencies with other subparts. As a consequence a local compilation would do. It is likely though that dependencies would develop across modules. It would then be unconvenient to have to link explicitly and provide paths for every possible module being used. In this case a partial install is useful by copying libraries and header files into a common iCub-wide directory. As a last option, when the system is ready to be deployed a possibly used by many users (perhaps connected to the same hardware platform) a system install is required. This typically copies the library, executables, and header files in some common place (e.g. /usr/local on Linux).
Our code is going to be multi-platform. We have decided to support Windows, Linux, and MacOS. The controller of the iCub is likely to run on multiple machines in parallel; consequently several

compilation might reside on the same directory tree. This is accomplished by defining a subdirectory for each compilation: i.e. libraries will be stored in places like "`lib/linux`" or "`lib/winnt`". The same applies to binaries.
The valid directory names for the various OSs are:

- winnt
- linux
- darwin

We need then to revise our previous table by adding a few directories to allow the install (not the system install!) on the current copy of the repository. It is useful to have an environment variable pointing to this root directory (the same place where we've done the checkout). We have chosen to use ICUB_ROOT which can be set for example by the following command:

```
export ICUB_ROOT=/usr/src/iCub
```

which throughout this manual will be referred as "`$ICUB_ROOT`". Also, in case you are planning the system-wide install another environment variable should be defined. This is ICUB_INSTALL which can be defined for example by the following command:

```
export ICUB_INSTALL=/usr/local/iCub
```

which will be also referred as "`$ICUB_INSTALL`".
Consequently, we need also the following directories for the local install under the $ICUB_ROOT directory:

Directory name	Description
include/iCub	header files
lib	with subdirectories for different OSs
bin	executables, with subdirectories for different OSs

Section 6 describes these different istallation and compilation options in details.

5.1 The logpolar module

We preferred to explain the organization of the code by providing an example of a self contained static library that converts images back and forth from rectangular to logpolar. The logpolar mapping is a model of the distribution of photoreceptors in the human retina. For a detailed manual of this module, please, check the documentation provided with the module (Doxygen & LaTeX). The table next shows the organization within the directory called "`logpolar`":

Directory name	Description
src	source code and makefile's
doc	documentation and manuals
include/iCub	header files
tools	application code

the home directory contains also the Doxygen file for building the reference manual and the main Makefile. Compilation would happen locally (see Makefile) and the results would be stored in "`obj/`" within a subdirectory specific to your operating system. This module builds a static library called:

`lib_iCub_logpolar.*`

where the * stands for the library extension on your system (e.g. .lib). On a local compilation (distinct from the system install) libraries are copied into a lib directory (in this case "`logpolar/lib/`") under a directory specific to your operating system (e.g. linux).
For more information please have a look at the "`doc`" directory and at the Doxygen manual pages. Also, the "`logp_libtest`" under "`tools`" contains an example that loads a rectangular image and performs the logpolar sampling and back to rectangular (remapping) performing the color reconstruction simultaneously. In addition, since several digital cameras have a Bayer pattern output, instead of the more traditional RGB, functions to reconstruct the RGB color from the Bayer battern have been implemented directly in the library.

6 Makefiles

In the example, several Makefile templates have been introduced. While it is not mandatory to use them, they are provided in the repository to ease the smooth integration of different modules. All Makefile's allow the following commands:

- clean: to clean the previous compilation object files;
- default: to build the current directory, possibly recursively; if you just type "`make`" it would do;
- install: to install locally, which means starting from "`$ICUB_ROOT`";
- sysinstall: to install system-wide to the local machine, which means copying files into "`$ICUB_INSTALL`";

We have already mentioned the role of the different install options. In particular by typing "`make`", compilation would start and the libraries and executables will be stored in a subdirectory of the current module (./lib and ./bin respectively). The templates are found in $ICUB_ROOT/conf. They are typically included in the main makefile's rather than used directly. In particular:

- Makefile.def: this template contains some definitions that are useful for compiling under the different OSes. For example, it tests the shell and sets the value of the $OS variable;
- Makefile.plain.template: this file can be included and used to compile recursively without any install. It simply checks a list of directories for additional makefile's and runs them;
- Makefile.src.template: this makefile can be included to compile static libraries following the iCub format presented in this document. It also contains the instructions for installing either locally or system-wide;
- Makefile.recursive.template: this is another makefile similar to the plain one for recursive compilation. It is different from the plain one in that it installs also from the current module directory;
- Makefile.tool.template: this makefile can be included to provide the instructions for compiling an executable and install it appropriately.

Option name	Description
CFAST = -g	compile with debug on, default off
MYLIB = ...	add a list of libraries to be linked

All makefile's accept the following compilation options:
For example:

```
$ make CFAST=-g
```

would compile with the debugging information into the generated code.

7 Compiling on Windows

Compilation on Windows is performed through the Microsoft Visual Studio 6.0. There are two projects to be used for compiling the library first and then the example as shown below for Linux. The project contains both the configurations for compiling locally (no install) or to perform the installation on the $ICUB_ROOT directory. The configuration for installing globally is not implemented yet.
As an example, starting the "`build`" command would display something like this on the output window:

```
-------Configuration: RC_LogPolar - Win32 Release Install--------
Compiling...
RC_DIST_FB_logpolar_mapper.cpp
Creating library...
Installing...
..\include\iCub\RC_DIST_FB_logpolar_mapper.h
        1 file(s) copied.
..\lib\winnt\lib_iCub_logpolar.lib
        1 file(s) copied.

lib_iCub_logpolar.lib - 0 error(s), 0 warning(s)
```

On Windows, there are always two different libraries generated whether the debug option is on. The version with the debug "`on`" has a trailing lowercase "d" in the filename. The extension is "lib" which is standard in Windows. Libraries are better static (not DLL) since remoting would be easier: i.e. in case the applications have to be run on a remote machine only the executables has to be physically moved there.
A similar project exists for compiling the test applications: have a look into the "`tools`" directory for a VisualStudio workspace (.dsw). Also in this case, there is the option to perform the installation. The syntax for executing the test application is exactly the same as in Linux (see below in section 8). As for Linux, you should take care of setting the proper path for running the application directly from the command window.

8 A short tutorial

Compiling the logpolar example is fairly straightforward provided the environment is set as mentioned above. In particular, once more, make sure you have defined the $ICUB_ROOT environment variable. This is required by the makefile provided. It would be convenient to add the directory of the binaries to the PATH environment variable (e.g. $ICUB_ROOT/-bin/linux) so that you can run the iCub applications just by typing their names at the command line. To start, then, just type:

```
$ cd $ICUB_ROOT
```

which brings you to the iCub software directory. If you "ls" there you should see something like this:

```
drwxr-xr-x  10 pasa users 4096 2005-07-08 01:48 .
drwxr-xr-x   8 root root  4096 2005-07-09 01:24 ..
-rw-r--r--   1 pasa users   70 2005-05-26 12:35 AUTHORS
drwxr-xr-x   6 pasa users 4096 2005-07-08 01:48 bin
drwxr-xr-x   3 pasa users 4096 2005-07-09 02:15 conf
-rw-r--r--   1 pasa users  313 2005-05-26 12:35 COPYING
drwxr-xr-x   2 pasa users 4096 2005-07-08 01:48 CVS
drwxr-xr-x   4 pasa users 4096 2005-07-08 01:48 doc
drwxr-xr-x   4 pasa users 4096 2005-07-08 01:48 include
-rw-r--r--   1 pasa users   50 2005-05-26 12:35 INSTALL
drwxr-xr-x   6 pasa users 4096 2005-07-08 01:48 lib
drwxr-xr-x   3 pasa users 4096 2005-07-08 01:48 license
-rw-r--r--   1 pasa users   93 2005-05-26 12:35 README
drwxr-xr-x   4 pasa users 4096 2005-07-08 14:11 src
-rw-r--r--   1 pasa users   16 2005-05-26 12:35 TODO
```

which should resemble the directory structure we described earlier. To start compilation then just type:

```
$ cd src/logpolar
$ make
```

Your terminal should display something like this:

```
make[1]: Entering directory '/usr/src/iCub/src/logpolar/src'
mkdir -p ../obj/linux
g++ -D_REENTRANT -O3 -I../include -I/usr/src/iCub/include/linux
-fexceptions -pipe -Wpointer-arith -Wno-uninitialized  -D__LINUX__
-DICUB_OS_CONFIG=LINUX -I/usr/include/g++/ -I/usr/src/iCub/include/
-DACE_NDEBUG -DACE_USE_RCSID=0 -DACE_HAS_EXCEPTIONS -D__ACE_INLINE__
-DACE_HAS_ACE_TOKEN -DACE_HAS_ACE_SVCCONF -DACE_AS_STATIC_LIBS
-c RC_DIST_FB_logpolar_mapper.cpp -o
../obj/linux/RC_DIST_FB_logpolar_mapper.o
mkdir -p ../lib/linux
Making lib_iCub_logpolar.a for ../obj/linux/RC_DIST_FB_logpolar_mapper.o
```

```
ar rv ../lib/linux/lib_iCub_logpolar.a
../obj/linux/RC_DIST_FB_logpolar_mapper.o
ar: creating ../lib/linux/lib_iCub_logpolar.a
a - ../obj/linux/RC_DIST_FB_logpolar_mapper.o
make[1]: Leaving directory '/usr/src/iCub/src/logpolar/src'
make[1]: Entering directory '/usr/src/iCub/src/logpolar/tools'
make[2]: Entering directory '/usr/src/iCub/src/logpolar/tools/logp_libtest'
mkdir -p obj/linux
g++ -O3  -I/usr/src/iCub/include/linux -I/usr/src/iCub/include -I.
-fexceptions -pipe
-Wpointer-arith -Wno-uninitialized  -D__LINUX__ -DICUB_OS_CONFIG=LINUX
-I/usr/include/g++/ -I/usr/src/iCub/include/ -DACE_NDEBUG -DACE_USE_RCSID=0
-DACE_HAS_EXCEPTIONS -D__ACE_INLINE__ -DACE_HAS_ACE_TOKEN
-DACE_HAS_ACE_SVCCONF -DACE_AS_STATIC_LIBS -c RC_DIST_FB_logp_test.cpp
-o obj/linux/RC_DIST_FB_logp_test.o
mkdir -p ../bin/linux
g++ -O3  -I/usr/src/iCub/include/linux -I/usr/src/iCub/include -I.
-fexceptions -pipe
-Wpointer-arith -Wno-uninitialized  -D__LINUX__ -DICUB_OS_CONFIG=LINUX
-I/usr/include/g++/ -I/usr/src/iCub/include/ -DACE_NDEBUG -DACE_USE_RCSID=0
-DACE_HAS_EXCEPTIONS -D__ACE_INLINE__ -DACE_HAS_ACE_TOKEN
-DACE_HAS_ACE_SVCCONF -DACE_AS_STATIC_LIBS obj/linux/RC_DIST_FB_logp_test.o
-o ../bin/linux/logp_libtest
-L/usr/src/iCub/lib/linux ../../lib/linux/lib_iCub_logpolar.a -lpthread -ldl
make[2]: Leaving directory '/usr/src/iCub/src/logpolar/tools/logp_libtest'
make[1]: Leaving directory '/usr/src/iCub/src/logpolar/tools'
```

where you should recognize the compilation of the static library and subsequently of the library test application. You might want then to install them locally by typing:

```
    $ make install
```

with the following result:

```
Installing lib_iCub_logpolar.a to /usr/src/iCub/lib/linux
mkdir -p /usr/src/iCub/include/iCub
Installing RC_DIST_FB_logpolar_mapper.h to /usr/src/iCub/include/iCub
make[1]: Entering directory '/usr/src/iCub/src/logpolar/src'
mkdir -p /usr/src/iCub/lib/linux
mkdir -p /usr/src/iCub/include/iCub
cp ../lib/linux/lib_iCub_logpolar.a /usr/src/iCub/lib/linux
cp -f ../include/iCub/*.h /usr/src/iCub/include/iCub
make[1]: Leaving directory '/usr/src/iCub/src/logpolar/src'
make[1]: Entering directory '/usr/src/iCub/src/logpolar/tools'
make[2]: Entering directory '/usr/src/iCub/src/logpolar/tools/logp_libtest'
mkdir -p /usr/src/iCub/bin/linux
cp ../bin/linux/logp_libtest /usr/src/iCub/bin/linux
make[2]: Leaving directory '/usr/src/iCub/src/logpolar/tools/logp_libtest'
make[1]: Leaving directory '/usr/src/iCub/src/logpolar/tools'
```

which finally copies everything to the right place. For example the executable is copied into `\$ICUB\_ROOT/bin/linux` in this case. To test the application, change directory to `./tools/logp\_libtest` and run:

```
$ logp_libtest imagebayer.bmp
```

which runs for a while generating several examples of logpolar and remapped images. In an ideal world you should get the following response:

```
logp_libtest: loading bitmap
logp_libtest: reconstructing color from bayer pattern
logp_libtest: building a logpolar image, bayer pattern, no color reconstruction
logp_libtest: reconstructing logpolar color
logp_libtest: building color logpolar image
logp_libtest: remapping from logpolar to rectangular
```

Note that this example is not supposed to run in realtime since it builds the conversion tables on the fly, while in the general case they can be built once and reused. In the same directory you will see several test images of the logpolar conversion for you to see.

9 Compilation support

The iCub servers will provide one machine per OS with ssh access to test compilation in a standard environment (Linux, Windows, MacOS). At the moment of writing we have a linux box at *hydra.lira.dist.unige.it* for this purpose. When this type of support will be fully available we will advertize it appropriately.

10 Indent utility

You can use "`indent`" to properly indent your code. The following command seems to work well with respect to the specifications of the iCub code:

```
$ indent FILENAME -bli0 -ts4 -nut -i4
```

It is not perhaps fully compliant yet.

11 Troubleshooting

Please contact Giorgio Metta at giorgio.metta@robotcub.org had you to encounter any problem, difficulty or a plain bug in either this manual or the CVS repository. The logpolar code was developed by Fabio Berton (fberton@dist.unige.it) and he is the one to blame (don't!).

Part III
Software Documentation and Coding Standards

12 Introduction

RobotCub is a collaborative project, both in its current developmental phase, when members of the consortium are working together to create this cognitive humanoid cub, and in its subsequent evolutionary phase, when the rest of the cognitive systems scientific community will join in the on-going enhancement of the iCub. Our goal is to create a system that can be easily adopted and modified by others. This in turn requires that it is presented as a consistent integrated body of work. Coding and documentation standards help us achieve this goal (although they are by no means sufficient to achieve the goal).

Ideally, all software is complemented both by informative in-line source code comments and by helpful external documents (such as tutorials, user manuals, and reference manuals). In the RobotCub project, we strongly encourage contributors to provide this type of external documentation, but it is not mandatory. Instead, we depend on well-documented in-line source code. We use the Doxygen tool [6] to extract source code comments and automatically create supporting documentation. The source code comments that are used by Doxygen must be identified by certain markup conventions and in the following we will distinguish between comments that are intended for automatic extraction and document creation by Doxygen and those that are intended only to enhance and explain the source code. However, we will provide guidelines for both types of comments. In addition, we will also provide a guide for formatting source code and a set of guidelines for programming style. We also provide a set of guidelines for naming conventions, including file-names.

In summary then, this deliverable provides a set of standards for:

1. Source code documentation (for extraction and document generation by Doxygen)
2. Source code comments (for explaining and illuminating the source code)
3. Formatting source code
4. Programming style
5. File name conventions

Because iCub software is written in a mix of conventional (imperative) C and object-oriented C++, we will to distinguish where appropriate between standards that apply to C source code, to C++ source code, and to both.

13 Contributing Standards

In creating these standards, we have drawn from several sources. These include:

- GNU Coding Standards [5];
- Java Code Conventions[4];

- C++ Coding Standard[2];
- The EPFL BIRG Coding Standards[1];
- The Doxygen User Manual[6].

The standards set out in this document represent an attempt to find a balance between specifying too much (and no one will read or use them) and specifying too little (and the desired consistency won't be achieved).

14 Languages, Compilers, and Operating Systems

iCub software should be written in either the C language or the C++ language. It should be possible to compile iCub software with both Microsoft Visual C++ 6.0 compiler and the GNU gcc and gpp compilers.

The RobotCub project supports Windows, Linux, and MacOS (Darwin).

15 File Names and Naming Conventions

15.1 Roots of File Names

A naming convention has already been established for documents related to the mechanical engineering parts of the iCub (see `RC_TLR_100_D_03_00_coding_standards.pdf`). The standard for naming software files and related documentation is similar to this convention but eliminates fields in the filename that are unnecessary in this context and adds two extra fields.

The syntax of a file name for software files and related documentation is defined as follows.

```
<filename>    ::= RC_<Group>_<Author>_<Project>_<Description>.<ext>

<Group>       ::= DIST | TLR | EPFL ...

<Author>      ::= XX          // author initials: two characters

<Project>     ::= <string>    // string that is descriptive of the overall
                              // purpose of the system; no underscores
                              // but dots (.) allowed

<Description> ::= <string>    // string that is descriptive of the
                              // function or content of the file;
                              // underscores allowed

<ext>        ::= XXX         // e.g. c, cpp, h, doc, tex, txt, pdf, ...
                              // see next section
```

The `<Project>` field should be used for all files associated with a given purpose, function, or system. For example: `ControlInterface`, `DynamicalModel`, `Standards`, or `Vision`. Each author in each Group (*i.e.* institute) is free to identify this field. This allows authors to create their own unique file names without having to check with a central database to see if that filename has been allocated. These file names will also be reasonably short.

Note that this naming convention will not apply to files that are not a direct result of the RobotCub project. For example, the GNU General Public License to be made available on the RobotCub website will be named simply `gpl.txt` and not, say, `RC_DIST_DV_license_gpl.txt`.

Do not use "−" (hyphens) or " " (spaces) in filenames.

15.2 File Name Extensions

The following file name extensions should be used for source code files.

	C	C++
Source Code	`c`	`cpp`
Header	`h`	`h`

15.3 Common File Names

Frequently-used file names include the following.

File Name	Use
`README`	The preferred name for the file that summarizes the contents of a particular directory

16 File Organization

In general, there are three different types of source code file in any given project. These are the interface, implementation, and application files.

In the case of projects coded using the C++ lanaguage, the interface file is a header file with the class declaration (with prototype method declarations but no method implementations) and templates. The implementation file contains the source code for the implementation of each class method.

When using `inline` methods/functions, put the `inline` keyword in front of the method/function implementation but *not* in front of the prototype declaration in the interface file. Use `inline` methods/functions sparingly.

In the case of projects coded using the C language, the interface file is a header file that contains the function prototype declarations. The implementation file contains the source code of common functions that could be of general use. Such functions might eventually be placed in a library.

In both cases, the application file contains the source code for the particular project in hand. It `#includes` the interface file. The application file will contain, for example, the code for the graphic user interface and the `main` function.

File names for each type of file should have the following extensions.

	C	C++
Implementation	c	cpp
Interface	h	h
Application	c	cpp

17 File Structure

17.1 General Guidelines

Either three or four spaces should be used as the unit of indentation. Choose one standard and stick to it throughout the code.

Do not use tabs to indent text. If you are using Microsoft Visual C++ (Visual Studio) turn off the tabs option (go to `Option->Tabs` and click the `insert spaces` radio button; you can set the number of spaces to three or four here too).

Avoid lines longer than 80 characters, since they are not handled well by many terminals and tools.

When an expression will not fit on a single line, break it according to the following general principles.

- Break after a comma.
- Break before an operator.
- Align the new line with the beginning of the expression at the same level on the previous line.

For example, consider the following statements.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longName6;  // Good break

longName1 = longName2 * (longName3 + longName4
                         - longName5) + 4 * longName6;  // bad break: avoid
```

17.2 Structure for C Files

17.2.1 Declaration of External Functions

Declarations of external functions and functions to appear later in the source file should all go in one place near the beginning of the file (somewhere before the first function definition in the file), or else should go in a header file.

Do not put `extern` declarations inside functions.

17.3 Structure for C++ Files

17.4 Use of Guards for Header Files

Include files should protect against multiple inclusion through the use of macros that guard the file. Specifically, every include file should begin with the following:

```
#ifndef FILENAME_H
#define FILENAME_H

...  header file contents go here

#endif /* FILENAME_H */
```

In the above, you should replace `FILENAME` with the root of the name of the include file being guarded *e.g.* if the include file is `cognition.h` you would write the following:

```
#ifndef COGNITION_H
#define COGNITION_H

...  header file contents go here

#endif /* COGNITION_H */
```

18 Source Code Documentation

18.1 Introduction

Programs should have two kinds of comments: implementation comments and documentation comments.

Implementation comments are those which explain or clarify some aspect of the code. They are delimited by `/* ... */` and `//`.

Documentation comments are intended to be extracted automatically by the Doxygen tool to create either HTML or LaTeX documentation for the program. They are delimited by `/** ... */`. Documentation comments are meant to describe the specification of the code from an implementation-free perspective. They are intended to be read by developers who won't necessarily have the source code at hand. Thus, documentation comments should help a developer understand the usage of the program, rather than its implementation.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program.

Information about how the executable should be compiled and linked or in what directory it resides should not be included as a comment. This information should go in the `README` file.

All comments should be written in English.

18.2 Implementation Comments

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

18.2.1 Block Comments

Block comments are used to provide descriptions of files, methods, data structures, and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

```
/*
 * Here is a block comment.
 */
```

18.2.2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format.

```
if (condition) {

   /* Handle the condition. */

   ...
}
```

18.2.3 Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a segment of code, they should all be indented to the same level.

```
if (a == b) {
   return TRUE;                 /* special case */
}
else {
   return general_answer(a);    /* only works if a != b */
}
```

18.2.4 End-Of-Line Comments

The `//` comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments. However, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow.

```
if (foo > 1) {

   // look left
   ...
}
else {
   return false;  // need to explain why
}



//if (foo > 1) {
//
//   // look left
//   ...
//}
//else {
//   return false;  // need to explain why
//}
```

18.2.5 The First Block Comment

All source files should include a block comment that gives the copyright notice and GPL license, as follows.

```
/*
 * Copyright (C) <year> <name of author>, <author institute>
 * RobotCub Consortium, European Commission FP6 Project IST-004370
 * email:   <firstname.secondname>@robotcub.org
 * website: www.robotcub.org
 *
 * Permission is granted to copy, distribute, and/or modify this program
 * under the terms of the GNU General Public License, version 2
 * or any later version published by the Free Software Foundation.
 *
 * A copy of the license can be found at
 * http://www.robotcub.org/icub/license/gpl.txt
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 * See the GNU General Public License for more details.
 */
```

This text is not included in a documentation comment (see below) because we don't want it to be extracted into the documentation by Doxygen. The documentation will contain its own GNU FDL license.

The comment should be placed at or close to the beginning of the file.

18.3 Documentation Comments

18.3.1 Preliminaries

Documentation comments describe classes, constructors, destructors, methods, members, and functions. The Doxygen documentation system [6] is used to extract the documentation comments and create external documentation in HTML or LaTeX. Although Doxygen supports several documentation formats, we will stick to the Javadoc format as it is widely-accepted and it facilitates visually-pleasing and unobstrusive comments.

Each documentation comment is set inside the comment delimiters `/** ... */`. Within this comment, several keywords are used to flag specific types of information (*e.g.* `@param`, `@see`, and `@return`). We will treat each of these below by way of example.

Documentation comments are placed in front of a declaration or definition. Although Doxygen allows documentation comments to be placed in other places, such as after a declaration, in another location, or in another file, we will stick to the convention that documentation comments are placed directly in front of a declaration or definition.

Note that blank lines are treated as paragraph separators and the resulting documentation will automatically have a new paragraph whenever a blank line is encountered in a documentation comment.

18.3.2 Brief and Detailed Descriptions

For each code item (class, constructor, destructor, method, member, and function) there are two types of descriptions, which together form the documentation: a brief description and detailed description. Both should be provided. Having more than one brief or detailed description is not allowed.

As the name suggests, a brief description is a short one-liner, whereas the detailed description provides longer more detailed documentation.

As noted above, we use the JavaDoc style so that the brief description is automatically taken from the first line of the comment block and it is terminated by the first dot followed by a space or new line. For example:

```
/** Brief description which ends at this dot. Details follow
 * here.
 */
```

If there is one brief description before a declaration and one before a definition of a code item, only the one before the declaration will be used.

If the same situation occurs for a detailed description, the opposite applies: the one before the definition is used and the one before the declaration will be ignored.

In short, brief descriptions before declarations have precedence over brief descriptions before definitions; detailed descriptions before definitions have precedence over detailed descriptions before declarations.

We recommend that you avoid confusion and simply put all documentation comments, i.e. *brief and detailed descriptions, before declarations in the interface (*`.h`*) file.*

Note that to use the JavaDoc style `JAVADOC_AUTOBRIEF` must be set to `YES` in the Doxygen configuration file.

18.3.3 The First Documentation Comment

All source files should begin with a documentation comment that lists the program or class name, version information, and date, as follows.

```
/** @file <filename> <one line to identify the nature of the file>
 *
 * Version information
 *
 * Date
 *
 */
```

18.3.4 Documenting Classes

There should be one documentation comment per class or function. This comment should appear just before the declaration:

```
/**
 * A test class. A more elaborate class description.
 */

class Test {

   public:

   /**
    * An enum.
    * More detailed enum description.
    */

   enum Tenum {
      TVAL1, /**< enum value TVAL1 */
      TVAL2, /**< enum value TVAL2 */
      TVAL3  /**< enum value TVAL3 */
   };

   Tenum *enumPtr;  /**< enum pointer.  Details. */
   Tenum  enumVar;  /**< enum variable. Details. */

   /**
    * A constructor.
    * A more elaborate description of the constructor.
    */
   Test();

   /**
    * A destructor.
    * A more elaborate description of the destructor.
    */

   ~Test();

   /**
    * a normal member taking two arguments and returning an integer value.
    * @param a an integer argument.
    * @param s a constant character pointer.
    * @see Test()
    * @see ~Test()
    * @see testMeToo()
    * @see publicVar()
    * @return The test results
    */

   int testMe(int a, const char *s);

   /**
    * A pure virtual member.
    * @see testMe()
    * @param c1 the first argument.
    * @param c2 the second argument.
    */

   virtual void testMeToo(char c1,char c2) = 0;

   /**
    * a public variable.
    * Details.
    */

   int publicVar;

   /**
    * a function variable.
    * Details.
    */

   int (*handler)(int a,int b);
};
```

Doxygen also allows you to put the documentation of members (including global functions) in front of the definition. This way the detailed documentation can be placed in the source file (definition) instead of the header file (declaration). Recall the point we made above about the precedence of definition and declaration regarding brief and detailed descriptions, and the recommendation that you put all documentation comments in the header (*i.e.* interface) file.

Top-level classes are not indented but their members are. The first line of a documentation comment is not indented but subsequent documentation comment lines each have one space of indentation (to align the asterisks vertically). Members, including constructors and destructors, have three or four spaces for the first documentation comment line (depending on which indentation standard you are using) and five spaces thereafter.

If you need to give information about a class, method, member, or function that isn't appropriate for documentation, use an implementation block comment or single-line comment immediately *after* the declaration. For example, details about the implementation of a class should go in such an implementation block comment *following* the class statement, not in the class documentation comment.

Documentation comments should not be positioned inside a method or a constructor definition block, because Doxygen associates documentation comments with the first declaration *after* the comment.

Documentation comments should, as a bare minimum, state:

- What the function or method does.
- What arguments it is passed, their types, and their use.
- What arguments it returns, their types, and their use.
- What the return type is, if any, and what it signifies.

18.3.5 Putting Documentation after Members

If you want to document the members of a file, struct, union, class, or enum, and you want to put the documentation for these members inside the compound, it is sometimes desired to place the documentation block after the member instead of before. For this purpose you should put an additional < marker in the comment block. For example:

```
    int var; /**< Detailed description after the member */
```

Warning: These blocks can only be used to document members and parameters. They cannot be used to document files, classes, unions, structs, groups, namespaces and enums themselves.

18.3.6 Documenting Global Code Items

To document a member of a C++ class, you must also document the class itself. The same holds for namespaces. *To document a global C function, typedef, enum or preprocessor definition you must first document the file that contains it.* This causes a problem because you can't put a document comment 'in front' of a file. Doxygen allows code items to be documented by putting the document comment somewhere else but you must then identify the code item being documented with a *structural command*.

To document a global code item, such as a C function, you must document the file in which they are defined by putting a document comment with file structural command

```
/** @file */
```

in that file. Usually this will be a header file. Here is an example of a C header named `structcmd.h`.

```
/** @file structcmd.h  A documented header file ...
 * These are the functions ...
 */


/** Opens a file descriptor.
 * @param pathname The name of the descriptor.
 * @param flags Opening flags.
 */
int open(const char *,int);


/** Closes the file descriptor .
 * @param fd The descriptor to close.
 */
int close(int);


/** Writes \a count bytes from \a buf to the filedescriptor \a fd.
 * @param fd The descriptor to write to.
 * @param buf The data buffer to write.
 * @param count The number of bytes to write.
 */
size_t write(int,const char *, size_t);
```

19 Programming Style

19.1 Declarations

19.1.1 Number Per Line

One declaration per line is recommended since it encourages commenting:

```
   int level; // indentation level
   int size;  // size of table
```

is preferable to:

```
   int level, size;
```

Do not put different types on the same line:

```
int foo, fooarray[]; //WRONG!
```

19.1.2 Initialization

Initialize local variables where they are declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

19.2 Placement

Put declarations only at the beginning of blocks. A block is any code surrounded by curly braces { and }. Don't wait to declare variables until their first use. Ideally, declare all variables at the beginning of the method or function block.

```
void myMethod() {
   int int1 = 0; // beginning of method block

   if (condition) {
      int int2 = 0; // beginning of "if" block
      ...
   }
}
```

19.2.1 Class Declarations

The following formatting rules should be followed:

- No space between a method name and the parenthesis (starting its parameter list.
- The open brace { appears at the end of the same line as the declaration statement.
- The closing brace } starts a line by itself indented to match its corresponding opening statement.

```
class Sample {
   ...
}
```

- Methods are separated by a blank line.

19.3 Statements

19.3.1 Simple Statements

Each line should contain at most one statement. For example:

```
argv++;          // Correct
argc++;          // Correct
argv++; argc--; // AVOID!
```

19.3.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces `{ statements }`. See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

- Braces are used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

```
if (condition) {
   a = b;
}
else {
   a = c;
}
```

19.3.3 `return` Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. For example:

```
return;

return myDisk.size();

return TRUE;
```

19.3.4 `if`, `if-else`, `if else-if else` Statements

The `if-else` class of statements should have the following form:

```
if (condition) {
   statements;
}

if (condition) {
   statements;
} else {
   statements;
}

if (condition) {
   statements;
} else if (condition) {
   statements;
} else {
   statements;
}
```

Always use braces `{ }`, with `if` statements. Don't use

```
if (condition) //AVOID!
   statement;
```

19.3.5 `for` Statements

A `for` statement should have the following form:

```
for (initialization; condition; update) {
   statements;
}
```

19.3.6 `while` Statements

A `while` statement should have the following form:

```
while (condition) {
   statements;
}
```

19.3.7 `do-while` Statements

A `do-while` statement should have the following form:

```
do {
   statements;
} while (condition);
```

19.3.8 `switch` Statements

A `switch` statement should have the following form:

```
switch (condition) {
case ABC:
   statements;
   /* falls through */
case DEF:
   statements;
   break;
case XYZ:
   statements;
   break;
default:
   statements;
   break;
}
```

Every time a case falls through (*i.e.* when it doesn't include a `break` statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every switch statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if later another `case` is added.

19.4 Naming Conventions

19.4.1 C vs. C++

Naming conventions make programs more understandable by making them easier to read. Since iCub software uses both the C language and the C++ language, sometimes using the imperative programming and object-oriented programming paradigms separately, sometimes using them together, we will adopt two different naming conventions, one for C and the other for C++. The naming conventions for C++ are derived from the JavaDoc standards [4].

19.4.2 C++ Language Conventions

The following are the naming conventions for identifiers when using C++ and the object-oriented paradigm.

Identifier Type	Rules for Naming	Examples

Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized	`class ImageDisplay`
		`class MotorController`

Methods	Method names should be verbs, in mixed case with the first letter in lowercase, with the first letter of each internal word capitalized	`int grabImage()`
		`int setVelocity()`

Variables	variable names should be in mixed case with the first letter in lowercase, with the first letter of each internal word capitalized	`int    i;`
		`float  f;`
		`double pixelValue;`

Constants	The names of variables declared as constants should be all uppercase with words separated by underscores _	`const int WIDTH = 4;`
Type Names	Typedef names should use the same naming policy as that used for class names	`typedef uint16 ModuleType`

Enum Names	Enum names should use the same naming policy as that used for class names.
	Enum labels should should be all uppercase with words separated by underscores _
		`enum PinState {`
		`   PIN_OFF,`
		`   PIN_ON`
		`};`

19.4.3 C Language Conventions

The following are the naming conventions for identifies when using C and the imperative programming paradigm.

Identifier Type	Rules for Naming	Examples

Functions	Function names should be all lowercase with words separated by underscores _	`int  display_image()`
		`void set_motor_control()`

Variables	variable names should be all lowercase with words separated by underscores _
	of each internal word capitalized
		`int    i;`
		`float  f;`
		`double pixel_value;`

Constants	Constants should be all uppercase with words separated by underscores _	`#define WIDTH 4`
`#define` and Macros	`#define` and macro names should all uppercase with words separated by underscores _	`#define SUB(a,b) ( (a) - (b) )`

19.5 And Finally: Where To Put The Opening Brace {

There are two main conventions on where to put the opening brace of a block. In this document, we have adopted the JavaDoc convention and put the brace on the same line as the statement preceding the block. For example:

```
class Sample {
   ...
}


while (condition) {
   statements;
}
```

The second convention is to place the brace on the line below the statement preceding the block and it indent it to the same level. For example:

```
class Sample
{
   ...
}


while (condition)
{
   statements;
}
```

If you really hate the JavaDoc format, use the second format, but be consistent and stick to it throughout your code.

20 Programming Practice

20.1 C++ Language Conventions

20.1.1 Access to Data Members

Don't make any class data member public without good reason.

One example of appropriate public data member is the case where the class is essentially a data structure, with no behaviour. In other words, if you would have used a struct instead of a class, then it's appropriate to make the class's data members public.

20.2 C Language Conventions

Use the Standard C syntax for function definitions:

```
void example_function (int an_integer, long a_long, short a_short)
...
```

If the arguments don't fit on one line, split the line according to the rules in Section 20.2:

```
void example_function (int an_integer, long a_long, short a_short,
                       float a_float, double a_double)
...
```

20.3 General Issues

20.3.1 Conditional Compilation

Avoid the use of conditional compilation. If your code deals with different configuration options, use a conventional `if-else` construct. If the code associated with either clause is long, put it in a separate function. For example, please write:

```
if (HAS_FOO) {
   ...
}
else {
   ...
}
```

instead of:

```
#ifdef HAS_FOO
   ...
#else
   ...
#endif
```

20.3.2 Writing Robust Programs

Avoid arbitrary limits on the size or length of any data structure, including arrays, by allocating all data structures dynamically. Use `malloc` or `new` to create data-structures of the appropriate size. Remember to avoid memory leakage by always using `free` and `delete` to deallocate dynamically-created data-structures.

Check every call to `malloc` or `new` to see if it returned `NULL`.

You must expect `free` to alter the contents of the block that was freed. Never access a data structure after it has been freed.

If `malloc` fails in a non-interactive program, make that a fatal error. In an interactive program, it is better to abort the current command and return to the command reader loop.

When static storage is to be written during program execution, use explicit C or C++ code to initialize it. Reserve C initialize declarations for data that will not be changed. Consider the following two examples.

```
static int two = 2; // two will never alter its value
...
static int flag;
flag = TRUE;        // might also be FALSE
```

20.3.3 Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

20.3.4 Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read.

Do not use the assignment operator in a place where it can be easily confused with the equality operator.

```
if (c++ = d++) { // AVOID!
    ...
}
```

should be written as

```
if ((c++ = d++) != 0) {
    ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler.

```
   d = (a = b + c) + r; // AVOID!
```

should be written as

```
   a = b + c;
   d = a + r;
```

20.3.5 Parentheses

Use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others — you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d)      // AVOID!

if ((a == b) && (c == d)) // USE
```

20.3.6 Standards for Graphical Interfaces

When you write a program that provides a graphical user interface (GUI), you should use a cross-platform library. At the very least, it must possible to compile your GUI code for both a Window environment and a Linux environment. The FLTK GUI library [3] satisfies this requirement.

20.3.7 Error Messages

Error messages should look like this:

```
   function_name: error message
```

20.3.8 License Messages

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
<Program name and version>

Copyright (C) <year> <name of author>, <author institute>
RobotCub Consortium, European Commission FP6 Project IST-004370
email:   <firstname.secondname>@robotcub.org
website: www.robotcub.org

This program comes with ABSOLUTELY NO WARRANTY.
Permission is granted to copy, distribute, and/or modify this program
under the terms of the GNU General Public License, version 2
or any later version published by the Free Software Foundation;
see http://www.robotcub.org/icub/license/gpl.txt
```

References

[1] *Birg Coding Standards for C/C++*. http://birg.epfl.ch/page26861.html.

[2] *C++ Coding Standards*. http://www.possibility.com/Cpp/CppCodingStandard.html.

[3] *FLTK User Manual*. http://www.fltk.org.

[4] *Java Code Conventions*. http://java.sun.com/docs/codeconv/CodeConventions.pdf.

[5] R. Stallman *et al.* *GNU Coding Standards*. 2005. http://www.gnu.org/prep/standards/.

[6] D. van Heesch. *Doxygen User Manual*. 2005. http://www.doxygen.org.